OBJECTIVES:

The Practical Extraction and Report Language (PERL) was developed by Larry Wall. In Perl, Larry Wall invented a general-purpose tool which is at once a programming language and the mother of all filters. Perl is standard on Linux. However, it is free and executables are available for all UNIX flavors. For more details, log on to http://www.perl.com.

1. PERL Preliminaries:

A perl program runs in a special interpretive mode, the entire script is compiled internally in memory before being executed. Unlike other interpreted language like the shell and awk, script errors are generated before execution itself.

```
#!/usr/bin/perl
#Sample perl script to show use of variables & computation – simple.pl
print "Enter Your City Name: ";
$name = <STDIN>;
print "Enter a temperature in Centigrade: ";
$centigrade = <STDIN>;
$fahrenheit = $centigrade * 9 / 5 + 32;
print "The temperature of $name in Fahrenheit is $fahrenheit\n";
```

We used the interpreter line as the first line in all of our perl script.
Perl variables need the $ prefix both in the definition as well as in evaluation.
<STDIN> is a file-handle representing standard input.
Note that unlike C, print function does not use parenthesis.

Let's execute the above Perl script using following commands-

- First we need to use **chmod 777 script_name.pl** to make the script executable.
- Then use **perl script_name.pl** to execute the script.

Execution & Output:

- $chmod 777 simple.pl
- $perl simple.pl

```
Enter Your City Name: Belgaum[Enter]
Enter a Temperature in Centigrade: 38[Enter]
```

```
The temperature of Belgaum
in Fahrenheit is 100.4
```

2. THE chop FUNCTION: REMOVING THE LAST CHARACTER

Why did above perl script show the output in two lines? That's because it included the newline generated by [Enter] as part of $name. So $name is now actually **Belgaum\n.** In many instances we need to remove the last character – especially when it's a newline. This is done by the **chop** function which is used in the below program.

```
#!/usr/bin/perl
#Sample perl script to show use of chop function – simple.pl
print "Enter Your City Name: ";
$name = <STDIN>;
chop($name);                            #Removes newline character from $name

print "Enter a temperature in Centigrade: ";
chop( $centigrade = <STDIN> );          #Removes newline character from $centigrade

$fahrenheit = $centigrade * 9 / 5 + 32;

print "The temperature of $name in Fahrenheit is $fahrenheit\n";
```

Execution & Output:

- $chmod 777 simple.pl
- $perl simple.pl

```
Enter Your City Name: Belgaum[Enter]
Enter a Temperature in Centigrade: 38[Enter]
The temperature of Belgaum in Fahrenheit is 100.4
```

3. VARIABLES AND OPERATORS

Variables

- Perl variables have no type and need no initialization.
- Perl variables need the $ prefix both in the definition as well as in evaluation.
- Strings and numbers can be as large as the machine permits.
- The following are some of the variable attributes one should remember- when a string can be used for numeric computation or comparison, perl immediately converts it into a number.
- If a variable is undefined, it is assumed to be a null string and a null string is numerically zero. Incrementing an uninitialized variable returns 1:

  Example:

  ```
  $ perl -e '$x++; print("$x\n");'
  output: 1
  ```
- if the first character of a string is not numeric, the entire string becomes numerically equivalent to zero.

Operators

Perl uses the following set of operators –

- Assignment: =
- Arithmetic: +, -, *, /, %
- short-hand operators: ++, --
- Numeric Comparison: ==, !=, >, <, >=, <=
- String Comparison: eq, ne, lt, gt, ge, le
- Conditional Operators: ?, :
- Logical Operators: &&, ||, !
- Concatenation operator: .
- Repeat Operator: x

4. THE STRING HANDLING FUNCTIONS:

The Perl has the following set of string handling functions-

- length($str); - Returns the length of the given source string
- index($str, $char); - Returns index of the given character in the source string
- substr($str, index, 0) = ”abcd”; - Stuffs the $str with abcd without replacing any character.
- $x = substr($str, -3, 2); - Extracts two characters from the third position on the right
- reverse($str); - Reverse the characters in $str & returns the reversed string
- uc($str); - Converts to uppercase
- ucfirst($str); - Converts only first character of each word in $str to uppercase
- lc($str); - Converts to lowercase
- lcfirst($str); - Converts only first character of each word in $str to lowercase

5. SPECIFYING FILE NAMES IN COMMAND LINE

The PERL provides specific functions to open a file and then perform I/O operations on it.
The diamond operator, <>, is used for reading lines from a file.
For example,

- <STDIN> reads a line from standard input file- keyboard.
- reads a line from the filename specified as command line argument at execution.
- **Examples:**
- perl -e 'print while(<>)' dept.lst
- perl -e 'print <>' dept.lst
- perl -e 'print while(<>)' foo1 foo2 foo3 foo4
- perl -ne 'print' dept.lst

6. $_: THE DEFAULT VARIABLE

Perl assigns the line read from input to a special variable, $_; often called default variable.
Chop, <> and pattern matching operate on $_ by default.
We can use $_ explicitly in print when it is required. otherwise, print also operates on $_ by default.

7. CURRENT LINE NUMBER($.) $ THE RANGE OPERATOR(..)

- Perl stores current line number in special variable, $.
- You can use it to represent a line address and select line from anywhere:
- **Examples:**
- perl -ne 'print if($. < 4)' foo #Like head -n 3
- perl -ne 'print if($. > 7 && $. < 11)' foo #Like sed -n '8,10p'
- Perl also use range operator, .. (2 dots)
- **Examples:**
- perl -ne 'print if($. < 4)' foo #Same as above example
- perl -ne 'print if($. > 7 && $. < 11)' foo #Same as above example

8. LISTS AND ARRAYS

Perl has a large number of functions that can manipulate lists and arrays.
The following is an example of a list:
(“jan”, 1234, “how are you”, -45.78, dec)
A list need not contain data of the same type as shown in above example.
For above list to be usable, it needs to be assigned to a set of variables, as follows-
($mon1, $num, $str, $neg, $mon2) = (“jan”, 1234, “how are you”, -45.78, dec);
When the size of a list can be determined only at runtime, we need an array to hold the list.

Let's assign above list to array, named @list:
@list = (“Jan”, 1234, “how are you”, -45.78, “Dec”);

Array Element Assignment Examples:
@week = (“Mon”, “Tue”, “Wed”, “Thr”, “Fri”, “Sat”, “Sun”);
@month = (“Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, 'July”, “Aug”, “Sept”, “Oct”, “Nov”, Dec”);
@month[1,3..5,12] = (“Jan”, “Mar”, “Apr”, “May”, “Dec”); #Assigns to specific index
@day = (1..30); #Assigns 1 to 30 integers, using range operator.
Perl supports the **qw** (quoted) function that can make short work of array element assignment.
Array Element Assignment examples using **qw** function:
@week = qw/Mon Tue Wed Thr Fri Sat Sun/; #No commas and no quotes
@month = qw/Jan Feb Mar Apr May Jun July Aug Sept Oct Nov Dec/; #Same as above
@month[1,3..5,12] = qw/Jan Mar Apr May Dec/; #Assigns to specific index

Array features:

- Arrays in perl are not of fixed size; they grow and shrink dynamically as elements are added and deleted.
- Each element of array is accessed using $arr_name[n], where n is the index, which starts from zero.
- When used as rvalue of an assignment, @arr_name evaluates to the length of the array:
- $arr_length = @arr_name;
- The $# prefix to an array name signifies the last index of an array which is always one less than the size of the array:
- $last_index = $#arr_name;
- The $# mechanism can also be used to set the array to a specific size or delete all its elements.
- $#arr_name = 10; #Array size now 11
- $#arr_name = -1; #No elements
- An array can also be populated by the <> operator. Each line then becomes an element of the array:
- @line = <>; #Reads entire file from command line

Example: Perl script to illustrate all the array features

```
#!/usr/bin/perl

@days_between = ("Wed","Thr");
@days = ( Mon, Tue, @days_between, Fri );
@days[5,6] = qw/Sat Sun/;

$length = @days;
$last_index = $#days;

print "Array Current Length is : $length \n";
print "Array Last Subscript is : $last_index \n";
print "Array Current Elements Are : @days \n";
print "Third Element of an Array is : $days[2] \n";

$#days = 4;
```

```
$resize = @days;

print "Array New Length is : $resize \n";
```

Output:

```
Array Current Length is : 7
Array Last Subscript is : 6
Array Current Elements are : Mon Tue Wed Thr Fri Sat Sun
Third Element of an Array is : Wed
Array New Length is : 5
```

9. ARGV[ ]: Command Line Arguments

The perl also uses command line arguments which are stored in system array - @ARGV[ ];
The first argument is $ARGV[0];
The second argument is $ARGV[1]; and so on

Note that the command name itself is not stored in this element. It is held in another system variable, $0;

```
#!/usr/bin/perl
#PERL SCRIPT TO ILLUSTRATE COMMAND LINE ARGUMENT – leap.pl

die("You have not entered the year \n") if (@ARGV == 0);

$year = $ARGV[0];
$last2digits = substr($year, -2, 2);

if($last2digits eq "00")
{
        $yesorno = ($year % 400 == 0 ? "leap" : "not a leap");
}
else
{
        $yesorno = ($year % 4 == 0? "leap" : "not a leap");
}
print("$year is " . $yesorno . " year \n");
```

Execution & Output:

```
$ chmod 777 leap.pl
$ perl leap.pl
you have not entered the year
$perl leap.pl 2004
2004 is certainly a leap year
$perl leap.pl 1997
1997 is not a leap year
```

10. Modifying Array Contents (Array Functions)

Perl has a number of functions for manipulating the contents of an arrays.

The various functions used to modify array contents are as follows –

1. push(@arr, elements) - Adds new element/elements at end

```
Let @list = (10, 20)
push (@list, 30)
Now @list = (10, 20, 30)

OR
Let @list = (10, 20)
push (@list, (30, 40))
Now @list = (10, 20, 30, 40)
```

2. pop(@arr) - Removes last element

```
Let @list = (10, 20)
$last = pop (@list)
Now $last = 20 and @list = (10)
```

3. unshift(@arr, elements) - Adds new element/elements at beginning

```
Let @list = (10, 20)
unshift (@list, 30)
```

Now @list = (30, 10, 20)

OR

Let @list = (10, 20)

unshift (@list, (30, 40))

Now @list = (30, 40 10, 20)

4. shift(@arr) - Removes first element

Let @list = (10, 20)

$last = shift (@list)

Now $last = 20 and @list = (10)

- splice( ) - Performs all the above operations

11. foreach: LOOPING THROUGH A LIST

Perl provides an extremely useful foreach construnt to loop through a list.

The construct borrowed from the C shell has a very simple syntax:

```
foreach $var ( @arr )
{
        statements;
}
```

Functionally, foreach works like the for loop of the shell.

Each element of the array @arr is picked up and assigned to the variable $var.

The iteration is continued as many times as there are elements in the array.

Example:

```
#!/usr/bin/perl
#Perl script to calculate square root of some numbers using - foreach

print "The program you are running is $0 \n";
foreach $number (@ARGV)
{
        print("The square root of $number is " . sqrt($number) . "\n");
}
```

Output:

```
$perl sqrt.pl 4 9 16 25
The program you are running is sqrt.pl
The square root of 4 is 2
The square root of 9 is 3
The square root of 16 is 4
The square root of 25 is 5
```

12. split: SPLITTING INTO A LIST OR ARRAY

Split breaks up a line or expression into fields.

These fields are assigned either to variables or an array.

Here are the two syntaxes:

- Splitting into Variables

  ($var1, $var2, $var3, ...) = split(/sep/, strg);

- Splitting into an Array

  @arr = split(/sep/, strg);

where,

/sep/ is an field separator symbol & **strg** is an input line or expression to split function.

Example: Splitting into Variables

```
#!/usr/bin/perl
#Perl script name - split_var.pl
print "Enter Three Numbers: ";
chop( $numstring = <STDIN> );
die("Nothing Entered! \n") if ( $numstring eq " " );
( $fn, $sn, $ln ) = split(/ /, $numstring );
print("The last, second and first Numbers are $ln, $sn and $fn \n");
```

Output:

```
$perl split_var.pl
Enter Three Numbers: 123 456 789
The last, second and first Numbers are 789, 456 and 123
```

Example: Splitting into an Array

```
#!/usr/bin/perl
#Perl script name - split_arr.pl
print "Enter Your Full Name: ";
chop( $namestring = <STDIN> );
die("Nothing Entered! \n") if ( $namestring eq " " );
@namearr = split(/ /, $namestring );
print("Your First Name is: $namearr[0] \n");
print("Your Middle Initial is: $namearr[1] \n");
print("Your Last Name is: $namearr[2] \n");
```

Output:

```
$perl split_arr.pl
Enter Your Full Name: Mahesh G Huddar
Your First Name is: Mahesh
Your Middle Initial is: G
Your Last Name is: Huddar
```

13. join: JOINING A LIST

The join function acts in an opposite manner to split.

It combines its arguments into a single string and uses the delimiter as the first argument.

The remaining arguments could be either an array name or list of variables or strings to be joined.

Examples:

- @week_array = (“Mon”, “Tue”, “Wed”, “Thr”, “Fri”, “Sat”, “Sun”);
- $weekstring = join(“ “,@week_array);
- $weekstring = join(“ “, “Mon”, “Tue”, “Wed”, “Thr”, “Fri”, “Sat”, “Sun”);

14. dec2bin.pl: CONVERTING A DECIMAL NUMBER TO BINARY

```
#!/usr/bin/perl
#Perl script to convert each decimal number to binary - dec2bin.pl
foreach $number(@ARGV)
{
$original_number = $number;
until( $number == 0 )
{
$bit = $number % 2;
unshift( @bit_arr, $bit );
$number = int ( $number / 2 );
}
$binary_number = join("",@bit_arr);
print "The binary number of $original_number is $binary_number \n";
$#bit_arr = -1;
}
```

Execution & Output:

```
$chmod 777 dec2bin.pl
$perl dec2bin.pl 2 4 6 8
The binary number of 2 is 10
The binary number of 4 is 100
The binary number of 6 is 110
The binary number of 8 is 1000
```

15. ASSOCIATIVE ARRAYS

Perl also supports a hash or associative array.
It alternates the array subscripts(array index) and values in a series of strings.
When declaring the associative array, these strings are delimited by commas or the more friendly => notation;

Examples:

```
%region = ( “N”, “North”, “S”, “South”, “E”, “East”, “W”, “West” );
%region = ( “N”=>“North”, “S”=>“South”, “E”=>“East”, “W”=>“West” );
%age = ( “Smith” => 12, “John” => 10, “Ken” => 11, “Jenny” => 14 );
```

This array uses the % symbol to prefix the array name.
The array subscript(index) , which can also be a string, is enclosed within a pair of curly braces rather than [ ].
For instance, $region{ “N” } produces North, $age{ “Ken” } produces 11.

Associative array having two important functions –

- keys – it displays all the subscripts(indexes) of associative array.
- Values – it displays all the values present in associative array.

Example:

```
#!/usr/bin/perl
#Perl script to illustrate associative array - region.pl
%region = ( "N"=>"North", "S"=>"South", "E"=>"East", "W"=>"West" );

foreach $index (@ARGV)
{
print "The letter $index stands for $region{$index} \n";
}

@key_list = keys(%region);
print "The Subscripts(Indexes) are @key_list \n";

@value_list = values(%region);
print "The values are @value_list \n";
```

```
The letter N stands for North
The letter S stands for South
The letter E stands for East
The letter W stands for West
The Subscripts(Indexes) are S W N E
The values are South West North East
```

16. REGULAR EXPRESSIONS AND SUBSTITUTION

Perl offers a grand superset of all possible regular expressions found in the UNIX system.
Perl understand both basic and extended regular expressions (BRE and ERE) & has some of its own too.

The s and tr functions

- The s and tr functions handle all substitution in perl.
- The s function is used for substitution.
- The tr function is used for character translation.
- The operators match(=~) and negate (!~) are used to match regular expressions with variables.
- The s and tr functions also accept flags.

- The s accept g flag for global substitution.
- The tr accept flags like – c for complements and d for delete character.

Examples:

- s/:/-/g; #Sets $_ when used this way
- tr/a-z/A-Z/; #Translate lowercase to uppercase
- $line =~ s/:/-/g; # $line is reassigned
- $name =~ tr/a-z/A-Z/; # $name is reassigned

Identifying Whitespace, Digits and Words

perl offers some escaped characters to represent whitespace, digits and word boundaries. The below table shows RE sequences & characters used by perl

Symbols	Significance
\w	Matches a word character (same as [a-zA-Z0-9_])
\W	Doesn't match a word character (same as [^a-zA-Z0-9_])
\d	Matches a digits (same as [0-9])
\D	Doesn't match a digits (same as [^0-9])
\s	Matches a whitespace character
\S	Doesn't match a whitespace character
\b	Matches on word boundary
\B	Doesn't match on word boundary
.	Any character except new line
+	One or more previous character
*	Zero or more previous character
^	Matches at the beginning
$	Matches at the end

17. FILE HANDLING

- Perl provides low-level file handling functions.
- A file is opened for reading like this:
- open(INFILE, “/home/henry/mailbox”);
- where, INFILE here is a file-handle of the file mailbox.
- Once the file has been opened, functions that read and write the file will use the file-handle to access the file.
- A file-handle is similar to file descriptor.
- A file is opened for writing with shell like operators, > and >>.
- open(OUTFILE, “>dept.lst”); #
- open(OUTFILE, “>>dept.lst”); #
- the statement while(<FILEIN>) reads a line at a time from the file represented by the FILEIN file-handle and stores it in $_. Every time the <FILEIN> statement is executed, the next line is read.

Example:

- $_ = <FILEIN>; #Assign to $_
- print; #print uses $_ by default
- files are explicitly closed by using close function.
- Examples:
- close(INFILE);
- close(OUTFILE);

18. FILE TESTS

Perl has an elaborate system of file tests.

The following statements test some of the common file attributes-

```
$x = “dept.lst”; #Assign file name to $x
print “File $x is Readable \n” if -r $x; #To check read permission
print “File $x is Executable \n” if -x $x; #To check execute permission
print “File $x has non-zero size \n” if -s $x; #To check file is empty or not
print “File $x is Exist \n” if -e $x; #To check file exist or not
print “File $x is Text File \n” if -T $x; #To check Text file or not
print “File $x is Binary File \n” if -B $x; #To check Binary file or not
```

20. SUBROUTINES

Perl supports functions but calls them subroutines.

A subroutine is called by the & symbol followed by the subroutine name.

If the subroutine is defined without any formal parameters, perl uses the array @_ as the default.

Variables inside the subroutines should be declared with my to make them invisible in the calling program.

Examples:

```
#!/usr/bin/perl
#Perl script to demonstrate how to create and use subroutines

print "Enter First Number:";
chop($n1 = <STDIN> );

print "Enter Second Number:";
chop($n2 = <STDIN> );

$sum = &add($n1,$n2);                    #Here, $n1 and $n2 are actual parameters

print "Addition of $n1 and $n2 is: $sum \n";

sub add()                                #Here, no formal parameters required
{
        return (@_[0] + @_[1]);
}
```

Execution & Output:

```
$ chmod 777 subroutine.pl
$ perl subroutine.pl
Enter First Number : 10 [Enter]
Enter Second Number : 20 [Enter]
Addition of 10 and 20 is : 30
```

The process

Process basics:

A process is a time image of a file. It is an instance of running program. A process is said to be born, when program starts execution and remains alive as long as the program is active. After execution is complete a process is said to die.

A process also has a name, usually the name of the program being executed. When user execute the cat command, name of the process is cat. Process have attributes, some attributes of every process maintained by the kernel in memory in a separate structure called Process table. Process table is inode for the process.

Two important attributes of process are:

1) **The process-id (PID)**: A unique identification number allotted to the process by kernel when process is born. PID is needed to control the process.
2) **The parent PID (PPID)**: PID of the parent is also available as process attribute.

The shell process:

When a user log on to a system, a process is immediately set up by the kernel. This represents a UNIX command, which may be sh (Bourne shell), ksh (Korn shell), csh (C shell) or bash (bash shell). Any command that user enters actually the standard input to the shell. Shell process remains alive until user logs-out.

The shell's pathname is stored in $SHELL: Shell's PID is stored in $$ variable.

Parents and children:

Every process has parent. A parent itself is another process and a process born from it is said to be its child.

Eg:

When user enters a
command $cat file1

A cat process starts by the shell. So shell is the parent of cat process. cat is the child of shell. The ancestry of every process ultimately traced to the first process. i.e. setup when system is booted. Each process can have only one parent, but parent can many children.

Eg:

```
$ls | wc
```

Setup two process for two commands and both are children of shell.

Wait and not wait:

- A parent may wait for the child to die. The death is informed by the kernel. When a command is executed in a shell, the shell waits for the command to die before it returns the prompt.
- It may not wait for the child to die and may continue to spawn other process. Generally init process (PID1) will do the something. i.e. why the init is the parent of many process.

All commands don't setup process. Built in commands of the shell like pwd, type, cd, etc don't create process.

<u>ps (Process status) command</u>:

ps command displays the process attributes. By default ps displays the processes owned by the user running command.

<u>Eg</u>:

```
$ps
PID          TTY          TIME          CMD
291          console      0:00          bash
302          pts/1        0:00          ps
```

First column shows the PID, second column shows terminal (TTY) with which process is associated. Third column shows cumulative processor time (TIME) last column shows process name (cmd).

<u>ps options:</u>

<u>-f (full listing)</u>: to get the detailed listing, we can use –f option.

<u>Eg</u>:

```
             $ps –f
UID          PID    PPID        C      STIME        TTY          TIME         CMD
Sumit        367    291         0      12:35        console      0:00         vi abc
Sumit        291    1           0      10:50        console      0:00         -bash
```

It shows PID and PPID also. Shell has PPID 1, i.e. the init process the second process od the system. '-' near to bash indicates login shell. STIME is the time of the process started. UID is the user id. C indicates %CPU time.

TTY	-	Control terminal of the process.
TIME	-	Cumulative CPU time.
CMD	-	Name of the process.

<u>-u (Displaying process of a user)</u>: System admin can run ps with –u option to know the activity of particular user.

<u>Eg</u>:

```
$ps –u sumit
PID          TTY           TIME         CMD
378           ?             0.05         xsun
403          pts/3          0:00         bash
438          pts/3          0.00          ps
```

<u>-a (Displaying all user process)</u>: -a option lists the process of all user, but doesn't display system process.

<u>Eg</u>:

```
$ps –a
PID          TTY          TIME          CMD
662          pts/1        0:00:00        bash
705          pts/2        0:00:00         vi
1680         pts/3        0:00:00         ps
```

-e or –A option: Displays all the processes including system processes. Most of system processes not associated with any control terminal. They are invoked during system startup. Some of them starts when system goes to multiuser state.

Eg:

```
$ps –e
PID         TTY        TIME       CMD
  0          ?         0:01        sched         //take care of swapping.
  1          ?         0:00         init         //parent of all shells.
  2          ?         0:00       pageout        //part of the kernel.
  3          ?         4:36       fsflush        //part of the kernel.
3054        pts/2      0:00         bash
3058        pts/2      0.00          ps
```

System processes are easily identified by '?' in the tty column.

They generally don't have controlling terminal, i.e. standard input and standard output of these process are not connected to the terminal. These processes are called deamons, because they are called without a specific request from user.

Some daemons are system process, they are:

/pshed	-	Controls printing activity.
sendmail	-	handles incoming and outgoing mails.
inetd	-	needs to run FTP and TELNET.
cron	-	Schedules jobs.

-l option

Long listing of process attributes

Mechanism of process creation:

There are 3 distinct phases in creation of process. and uses 3 important system call or functions – fork, exec and wait.

- **fork**: Fork process is created with the fork() system call, which creates a copy of the process the invokes it. (for example: when user runs a command from the shell, for example: cat command, the shell first forks another shell process.)

  The process image is practically identical to that of calling process except some parameters like PID, PPID. When process is forked, the child gets a new PID. The forking mechanism is responsible for multiplication of processes in the system.

- **exec**: Forking creates a process, but it is not enough to run a new program, to do forked child need to overwrite its own image with the code and data of new program. This mechanism is called exec and the child process is said to exec a new program. No new process is created here. So PID and PPID of exec process remains unchanged.

- **wait**: The parent executes the wait system call to wait for the child process to complete. It picks up exit status of the child and continues with its other function.

  Eg:

When user enters a command, for example: cat from the shell. The shell first forks another shell process using fork() system call. The newly forked shell overlays itself with executable image of cat (exec) which then starts to run. The parent (the shell) waits for cat to terminate and then picks up the exit status of the child. Exit status is number returned by the child to kernel. When a process is forked, the child has a different PID and PPID from its parent. But it inherits many properties from parent.

Some of the important attributes that are inherited are:

- **real UID and real GID of the process:** These are the UID and GID of the user running program (not the UID and GID of owner of the file).

- **The effective UID and effective GID of the process**: These are generally same as the real UID and real GID, but when third bit of user permission field is set (bit is called set_user_id (SUID)), then effective UID and effective GID is the UID and GID of owner of the file.

  Eg:

```
                $ls –l /usr/bin/passwd
-rwsr-xr-x   1  root   shadow   68680   2013-03-05   10:02   /usr/bin/passwd
```

In the example, set_user_id (SUID) bit is set. So if kumar runs passwd command, real UID and real GID is the UID and GID of kumar, but effective UID and GID is the root and shadow.

- **Current directory.**

- **File descriptor of all files opened by parent**

- **Environment variable like HOME and PATH**.

Shell creation:

$$\begin{tikzcd}
\text{init} \arrow[r, "\text{fork-exec}"'] & \text{getty} \arrow[r, phantom, "\text{fork-exec}"'] & \text{login} \arrow[r, phantom, "\text{fork-exec}"'] & \text{shell}
\end{tikzcd}$$

When system moves to multiuser mode, init fork and exec's login program to verify login name and password. On successful login, login fork-exec's the process representing login shell. Repeated overlaying results in init becoming immediate ancestor of the shell.

Internal and External commands

From the process view point, the shell recognizes 3 types of commands:

- External commands
- Internal commands
- Shell scripts

- **External commands**: External commands have independent existence in the system. Eg: cat, ls, etc.

  The shell creates a process for each of these commands that it executes while remaining their parent.

- **Internal commands**: The shell has number of built in commands. Some of them (like cd, pwd, echo) don't generate process. Variable assignment (for example: x=5) also doesn't generate process.

- **Shell scripts**: The shell executes these scripts by invoking another shell, which then executes the commands listed in the script. The child shell becomes the parent of the commands that feature in the script.

Why directory change (cd command) can't be made in separate process:

Some commands are built in to the shell, because it would be either difficult or impossible to implement them as separate external commands. It is necessary for cd command not to spawn a child to achieve change of directory. If separate child process is created, then after cd has completed its run control revert to the parent and original directory would be restored. (Because, child process inherit parent's current directory as its current directory.)

Process states and ZOMBIES:

At any instant of time, a process is in particular state. A process after creation is in the runnable state before it actually runs, When process is in run, it will be in running state. While process is running, it may invoke disk I/O operation, it will waits for I/O complete, this state is called waiting or sleeping
state. The process woke up operation completes. A process can be suspended by using (ctrl-z)

key.

Process whose when I/O parents don't wait for their death moves to zombie state. When a process dies, it immediately moves to the zombie state. It remains in this state until parent picks up the child's exit status. A zombie is a harmless dead child, but user can't kill it. It is possible that parent itself die before the child dies, then child becomes orphan and the kernel makes init the parent of all orphan.

Running jobs in background:

A multitasking system lets a user to do more than one job at a time. But there can be only one job in the foreground, the rest of the jobs have to run in the background. There are two ways to run the jobs in background, they are:

1) & operator.
2) nohup command.

1) **&: No logging out**: The '&' is the shell's operator used to run process in the background. The parent in this doesn't wait for the child's death, for this terminate command line with '&'.

Eg:

```
$cat file1 &
550                                   //Job's PID.
```

The shell immediately returns a number, the PID of invoked command. The prompt is returned and shell is ready to accept another command even though the previous command has not been terminated yet. Background execution of job is useful feature using this user can send low-priority time consuming job to the background. '&' can run important jobs in foreground.

2) **Nohup: Log out safely**: Background jobs will be killed when a user logs out. It is because, when user logs out, his shell will be killed. So when parent is killed, children are also normally killed. The nohup (no hang up) command permits execution of the process even after the user has logged out.

Syntax:

```
$nohup command &
```

Eg:

```
$nohup cat file1 &
586
Sending output to nohup.out.
```

The shell returns PID and some shell displays the messages like sending outputs to nohup.out. So nohup sends standard output of the command to nohup.out, if output is not redirected to any file. When the user logs out, init takes over the parentage of any process running with nohup. In this way user can kill the parent without killing its child. If user wants to use nohup command in pipeline, then it is necessary to use nohup command at the beginning of each command in the pipeline.

Eg:

```
$nohup ls & | nohup wc &
```

nice: Job execution with low priority:

Processes in the UNIX system executed with equal priority. UNIX offers "nice" command which is used with '&' operator to reduce the priority of jobs. More important jobs can then have greater access to the system resource.

To run the job with lower priority, the command should be prefixed with "nice".

Eg:

```
$nice wc –l manual
          Or
$nice wc –l manual &
```

nice is built in command in cshell. nice values are system dependent and typically range from 1-19. A higher nice value implies a lower priority. If it is possible, specify the nice value explicitly using –n option.

Eg:

```
$nice –n 5 wc –l manual &                    //nice value increased by 5units.
```

Killing a process with signals:

The occurrence of events can be conveyed to a process by signals. Each signals is identified by a number. Because, the same signal number may represent two different signals on two different machines, signals are better represented using its symbolic names having SIG prefix.

If user presses the interrupt key, system sends SIGINT signal (numbered 2) to the process. Its default action is killing process. A process may also ignore a signal or execute some user defined code written to handle that signal. There are two signals that process can't ignore or run user defined code to handle it, they are SIGKILL and SIGSTOP. These two signals performs the default action associated with them.

kill: premature termination of process:

Kill command sends a signal, usually with the intension of killing one or more processes. Kill is an internal command in most shells. The command uses one or more PID as its arguments.

Eg:

```
$kill 105                                    //same as $kill -s TERM 105
```

Terminates the job having PID 105

```
$kill 121 122 125
```

Terminates the jobs having PID 121 122 and 123. If all these process having same parent, user can kill parent in order to kill all children. When user use nohup with a set of command and log out, then it is not possible to kill the parent, as init acquires their parentage. Then processes should be killed individually, because it is not possible to kill init.

<u>**Killing last background job**</u>:

$! Contains PID of last background job.

<u>Eg</u>:

```
$cat manual &
345
$kill $!                              // kills the cal command
```

<u>**Using kill with other signal**</u>:

By default kill uses SIGTERM signal (15) to terminate process. But some commands ignores this signal and continues execution normally. In that case, the process can be killed with SIGKILL signal (9)

<u>**Syntax**</u>:

```
kill –s KILL
        121 or
kill -9 121
```

A simple kill command (with TERM) won't kill login shell. Login shell can be killed using following signal.

```
$kill -9   $$                         //kills login shell
$kill –s KILL 0                       //kills all processes including the login shell
```

To view the list of all signal names and number, -l option can be used with kill.

```
$kill –l
```

Or we can view the file /usr/include/sys/signal.h

<u>**Job control**</u>: A job is name given to a group of processes. Job control facility is available to control the execution of job. Using job control following tasks can be performed:

- Relegate a job to the background (bg).
- Bring it back to foreground (fg).
- List the active jobs (jobs).
- Suspend the foreground job (ctrl-z).
- Kill a job (kill).

We can suspend the job using (ctrl-z), then we can send it to background using bg command. We can use jobs command to view the status.

<u>Eg</u>:

```
$jobs
[3] + Running           wc –l *.c &
[1] - Running           cat *.c &
[2]  Running            cat *.o &
```

To bring the current (most recent) job to foreground command
is fg

This brings wc command to foreground. fg and bg command can be used with job number.

<u>Eg</u>:

```
$fg %1        →     Brings first job to foreground.
$fg %wc       →     Brings wc job to foreground.
$bg           →     Sends second job to background
$bg %?perm    →     Sends to background job containing string perm
```

We can use kill command to kill the job.

Eg:

```
$kill %1        →       Kills first background job.
```

at and batch commands (Excute later)

at: one time execution: at takes the argument, the time, the job to be executed.

Eg:

```
$at 14:08
at > cat newfile
```

Then job goes to queue and at 2:08 p.m. the command will be executed. The standard output and standard error is mailed to user.

Eg:

```
$at 15:08
$cat newfile > log
```

Runs the cat command at 3:08 p.m. and stores output in the file log. -f option can be used to take commands from a file.

-m option mails the user after job completion.

Eg:

```
$at 15                          //executes job at 3p.m.
$at 5pm                         //executes job at 5p.m.
$at noon                        //executes job at 12:00 hours today.
$at now+1year                   //executes job at current time after 1 year.
$at 3:08pm+1day                 //executes job at 3:08p.m. tomorrow.
$at 9am                         //executes job at 9a.m.
$at 15:08 December 18, 2021
```

Jobs can be listed using at –l command and removed using at –r or atrm command.

batch: execute in batch queue: Batch also schedules jobs for later execution, but jobs are executed as soon as the system load permits.

Eg:

```
$batch < newfile.sh
```

Above command executes the newfile.sh when system load permits.

cron: Running job periodically:

cron execute the program at regular interval. Cron daemon is always running, it wakes up every minute and looks in a control file /var/spool/cron/crontabs for instructions that should be performed in that instant. After execution it goes back to sleep and it wakes next minute. A user can also place crontab file in the crontabs directory.

Eg: abc can place crontab file in /var/spool/cron/crontabs/abc

Entry in the file can look like as follows:

1st	2nd	3rd	4th	5th	6th
00-10	17	*	3,5,7	5	cat *.c

Each line contains set 6 fields separated by space.

The first field (legal value 00-59) specifies number of minutes after the hour, when command is to be executed. In the above example, schedules the execution every minutes in the first 10 minutes of the hour.

The second field indicates the hour in 24 hours format (legal values 1-24). In the above example 17 means 5pm.

Third field indicates day of the month (legal values 1 to 31). * indicates everyday.

Fourth field indicates month (legal values 1-12).

Fifth field indicates day of the week (legal value 0-6). In the example 5 indicates Friday.

Sixth field indicates the command to be executed. So command will be executed every minute in the first 10 minute after 5pm. Every Friday of the months march, may, august. (Even though 3rd field uses * to indicate execute every day, fifth field overrides the entry and limits execution to every Friday.)

Eg:

```
* * * * * cat *.c
```

Above entry executes the command everyday in every minute.

Eg:

```
30      21      *       *       *       cat *.c
```

Above entry executes the command at 9:30pm everyday.

crontab: Creating a crontab file:

User can create their own crontab file. Steps are:

- Create a file and specify the entry.

  Eg:

  ```
  $vi cron.txt
  30      21      *       *       *       cat *.c
  ```

- After creating file, run the following command

  ```
  $crontab cron.txt
  ```

- If user abc runs the command, a file named abc will be created in /var/spod/cron/crontabs

$crontab –e option can be used to enter the cron command.

Contents of crontab file can be viewed using the command crontab –l.
Can remove the contents using crontab –r.

time: Timing processes:

We can view the time taken by the processes using time command. Eg:

```
$time sort file1
real 0m0.0625
user 0m0.0025
sys 0m0.0035
```

real time: Clock elapsed time from invocation of the command until its termination.
user time: Time taken by the program in executing itself.
sys time: Time taken by the kernel in doing work on behalf of user process.

The sum of user time and sys time actually represents CPU time.